תרגילים בעצים: אור יאנקו

בכל סעיף אפשר להשתמש בפונרציות של סעיפים הקודמים,ישנו לכל סעיף MAIN משלו

סעיף 1

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <malloc.h>

#include "queue.h"

typedef int itemType;

typedef struct tree

{

itemType info;

struct tree\* left;

struct tree\* right;

} tree\_node, \* tree\_ptr;

//פונקציה המוסיפה איבר לעץ

void insertTree(tree\_ptr\* t, int key)

{

tree\_ptr q=NULL, p;

if (\*t == NULL)

{

p = (tree\_ptr)malloc(sizeof(struct tree));

p->info = key;

p->left = NULL;

p->right = NULL;

\*t = p;

}

else

{

p = \*t;

while (p != NULL && p->info != key)

{

q = p;

p = (key < p->info) ? p->left : p->right;

}

if (p == NULL)

{

p = (tree\_ptr)malloc(sizeof(struct tree));

p->info = key;

p->left = NULL;

p->right = NULL;

if (q->info > key)

q->left = p;

else

q->right = p;

}

}

}

//פונקציה המוצאת את גובה העץ

int height(struct tree\* node)

{

if (node == NULL)

return 0;

else

{

int lheight = height(node->left);

int rheight = height(node->right);

/\* use the larger one \*/

if (lheight > rheight)

return(lheight + 1);

else return(rheight + 1);

}

}

//פונקציה המדפיסה רמה מסוימת

void print\_given\_level(struct tree\* root, int level)

{

if (root == NULL)

return;

if (level == 1)

printf("%d ", root->info);

else if (level > 1)

{

print\_given\_level(root->left, level - 1);

print\_given\_level(root->right, level - 1);

}

}

//פונקציות אשר מדפיסות את ארבעת ההדפסות של עץ

void pre\_print(tree\_ptr t)

{

if (t != NULL)

{

printf("%d ", t->info);

pre\_print(t->left);

pre\_print(t->right);

}

}

void in\_print(tree\_ptr t)

{

if (t != NULL)

{

in\_print(t->left);

printf("%d ", t->info);

in\_print(t->right);

}

}

void post\_print(tree\_ptr t)

{

if (t != NULL)

{

post\_print(t->left);

post\_print(t->right);

printf("%d ", t->info);

}

}

void printLevelOrder(struct tree\* root)

{

int h = height(root);

int i;

for (i = 1; i <= h; i++) {

print\_given\_level(root, i);

}

}

//פונקציה אשר מנהלת איזו סוג הדפסה להדפיס

void printTree(tree\_ptr t)

{

int x = 0;

printf("press 1 to preOrder\npress 2 to inOrder \npress 3 to postOrder\npress 4 to level search\t");

scanf("%d", &x);

if (x == 1)

pre\_print(t);

else if (x == 2)

in\_print(t);

else if (x == 3)

post\_print(t);

else if (x == 4)

printLevelOrder(t);

printf("\n");

}

//פונקציה הבונה עץ ריק

tree\_ptr buildTree(void)

{

return NULL;

}

//פונקציה הבודקת האם מספר נמצא בעץ

int if\_in\_tree(tree\_ptr t,int num)

{

if (t == NULL)

return 0;

int l = if\_in\_tree(t->left,num);

int r = if\_in\_tree(t->right,num);

if (t->info == num)

return 1;

if (l == 0 || r == 0)

return 1;

return 0;

}

//פונקציה המחזירה ערך הכי קטן בעץ

struct tree\* minValueNode(struct tree\* node)

{

struct tree\* current = node;

/\* loop down to find the leftmost leaf \*/

while (current && current->left != NULL)

current = current->left;

return current;

}

//פונקציה המוחקת מספר מהעץ

struct tree\* delete\_num\_from\_tree(tree\_ptr root, int key)

{

if (root == NULL) return root;

if (key < root->info)

root->left = delete\_num\_from\_tree(root->left, key);

else if (key > root->info)

root->right = delete\_num\_from\_tree(root->right, key);

else

{

if (root->left == NULL)

{

struct tree\* temp = root->right;

free(root);

return temp;

}

else if (root->right == NULL)

{

struct tree\* temp = root->left;

free(root);

return temp;

}

struct tree\* temp = minValueNode(root->right);

root->info = temp->info;

root->right = delete\_num\_from\_tree(root->right, temp->info);

}

return root;

}

//פונקציה המחזירה צומץ של KEY

tree\_ptr find\_node(tree\_ptr t, int num)//flag should be 0 at the begining

{

if (t==NULL)

return NULL;

find\_node(t->left, num);

if (t->info == num)

return t;

return find\_node(t->right, num);

}

//פונקציה המחזירה את הצומת הבאה בסריקת איןאורדר

tree\_ptr find\_next\_in\_inOrder(tree\_ptr t, int num, tree\_ptr original\_t,queue\* q)

{

if (t == original\_t)

{

queue\_init(q);

}

if (t != NULL)

{

find\_next\_in\_inOrder(t->left,num,original\_t,q);

enter\_queue(q, t->info);

find\_next\_in\_inOrder(t->right,num,original\_t,q);

}

if (t == original\_t)

{

int deleted = -1;

while (q->head->info != num)

deleted = delete\_queue(q);

if (q == NULL)

return NULL;

deleted=delete\_queue(q);

if (q == NULL)

return NULL;

int x = delete\_queue(q);

tree\* w = find\_node(t, x);

return w;

}

}

/\*

//ex2---------------------------------------------------------------

void find\_max\_in\_level(struct node\* root, int level,int\* max)

{

if (root == NULL)

return;

if (level == 1)

\*max = root->info;

else if (level > 1)

{

if (level == height(root))

{

}

}

}

void print\_biggest\_in\_any\_level()

{

}

void scan\_str\_and\_create\_tree\_without\_dups()

{

int num;

tree\_ptr t = buildTree();

scanf("%d", &num);

while (num != 0)

{

if (find\_node(t, num))

{

insertTree(&t, num);

}

scanf("%d", &num);

}

//א

printLevelOrder(t);

//ב

}

\*/

void main()

{

tree\_ptr t = buildTree();

insertTree(&t, 2);

insertTree(&t, 4);

insertTree(&t, 39);

insertTree(&t, 67);

insertTree(&t, 98);

insertTree(&t, 7);

printTree(t);

queue q;

tree\_ptr r=find\_next\_in\_inOrder(t, 98,t,&q);

if(r)

printf("\n%d", r->info);

}

סעיף 2

//פונקצית עזר המחזירה את הרמה

int getLevelHelp(struct tree\* node,int data, int level)

{

if (node == NULL)

return 0;

if (node->info == data)

return level;

int downlevel = getLevelHelp(node->left,data, level + 1);

if (downlevel != 0)

return downlevel;

downlevel = getLevelHelp(node->right,

data, level + 1);

return downlevel;

}

//מחזירה את רמת הערך

int find\_Level(struct tree\* node, int data)

{

return getLevelHelp(node, data, 1);

}

//פונקציה המוצאת מספר הכי גדול בכל רמה

void find\_max\_in\_level(struct tree\* original ,struct tree\* t, int level,int\* max)

{

if (t == NULL)

{

return;

}

if (level == 1)

{

\*max = original->info;

return;

}

if (original == t)

\*max = 0;

find\_max\_in\_level(original, t->left, level, max);

find\_max\_in\_level(original, t->right, level, max);

if (level==find\_Level(original,t->info))

{

if (\*max < t->info)

\*max = t->info;

}

}

//פונקציה המדפיסה כל מספר הכי גדול לפי רמה

void print\_biggest\_in\_any\_level(tree \*t)

{

int h = height(t); int max;

for (int i = 1; i <= h; i++)

{

find\_max\_in\_level(t, t, i, &max);

printf("max in level %d is :%d\n", i, max);

}

}

//סעיף 2

void scan\_str\_and\_create\_tree\_without\_dups()

{

int num;

tree\_ptr t = buildTree();

scanf("%d", &num);

while (num != 0)

{

if (find\_node(t, num)==NULL)

{

insertTree(&t, num);

}

scanf("%d", &num);

}

//א

printLevelOrder(t);

//ב

printf("\n");

print\_biggest\_in\_any\_level(t);

}

void main()

{

scan\_str\_and\_create\_tree\_without\_dups();

}

סעיף 3

//מחזירה את מספר הצמתים של שורש

int num\_of\_tzmatim(tree\_ptr t)

{

if (t == NULL)

return 0;

return num\_of\_tzmatim(t->left) + num\_of\_tzmatim(t->right) + 1;

}

//מחזירה 0 אם מספר הצמתים בצד ימין גדול ממספר הצמתים בצד שמאל של כל צומת בעץ ו1 להפך

int if\_left\_bigger\_than\_right(tree\_ptr t)

{

if (t == NULL)

return 1;

if (t->left == NULL && t->right == NULL)

return 1;

if (num\_of\_tzmatim(t->left) < num\_of\_tzmatim(t->right))

{

return 0;

}

return if\_left\_bigger\_than\_right(t->left) && if\_left\_bigger\_than\_right(t->right);

}

//סעיף 3 עצמו

void scan\_and\_return\_1\_or\_0\_if\_num\_of\_tzmatim\_in\_left\_bigger\_than\_right()

{

int num;

tree\_ptr t = buildTree();

scanf("%d", &num);

while (num != 0)

{

if (find\_node(t, num) == NULL)

{

insertTree(&t, num);

}

scanf("%d", &num);

}

printf("%d\n", num\_of\_tzmatim(t));

int x=if\_left\_bigger\_than\_right(t);

if (x == 1)

printf("True, in every tzomet there are more tzmatim in the left ;)\n");

else

printf("False, in every tzomet there aren't more tzmatim in the left :(\n");

}

void main()

{

scan\_and\_return\_1\_or\_0\_if\_num\_of\_tzmatim\_in\_left\_bigger\_than\_right();

}

סעיף 4

//פונקציה אשר מוסיפה למערך בכל אינדקס את מספר בצמתים בתאותה רמה של האינדקס

void add\_to\_arr\_from\_tree(tree\_ptr original, tree\_ptr t, int arr[])

{

if (t == NULL)

return;

int level = find\_Level(original,t->info);

arr[level-1]++;

add\_to\_arr\_from\_tree(original,t->left, arr);

add\_to\_arr\_from\_tree(original,t->right, arr);

}

//פונקציה המחזירה את המספר הגדול מבין 2 מספרים

int max(int x, int y)

{

return (x > y ? x : y);

}

void scan\_and\_print\_num\_of\_level\_with\_the\_most\_tzmatim()

{

int num;

tree\_ptr t = buildTree();

int arr[4]{ 0,0,0,0};

scanf("%d", &num);

while (num != 0)

{

if (find\_node(t, num) == NULL)

{

insertTree(&t, num);

}

scanf("%d", &num);

}

add\_to\_arr\_from\_tree(t, t,arr);

int xMax = 0;

xMax = max(arr[0], arr[1]);

xMax = max(arr[2], xMax);

xMax = max(arr[3], xMax);

int x1=0, x2=0, x3=0, x4=0;

int i = 0;

if (xMax == 0)

{

printf("NULL tree");

return;

}

while (xMax != arr[i])

i++;

x1 = i;

i++;

for (; i < 4; i++)

{

if (arr[i] == xMax)

x2 = i;

}

for (; i < 4; i++)

{

if (arr[i] == xMax)

x3 = i;

}

for (; i < 3; i++)

{

if (arr[i] == xMax)

x4 = i;

}

if (x4 != 0)

{

printf("num of tzmatim in all levels is the same = 1");

return;

}

else if (x3 != 0)

{

printf("num of tzmatim in levels %d %d %d is the same = %d",x1+1,x2+1,x3+1,arr[x1]);

return;

}

else if (x2 != 0)

{

printf("num of tzmatim in levels %d %d is the same = %d", x1+1, x2+1, arr[x1]);

return;

}

else

printf("num of tzmatim in levels %d is the same = %d", x1+1, arr[x1]);

}

void main()

{

scan\_and\_print\_num\_of\_level\_with\_the\_most\_tzmatim();

}

סעיף 5

void print\_ex5(tree\_ptr check, int level, tree\_ptr t, int OgLevel) {

if (check == NULL)

return;

if (level == 1) {

if (height(t) - height(check) == OgLevel - 1)

printf("%d ,", check->info);

}

else if (level > 1)

{

print\_ex5(check->left, level - 1, t, OgLevel);

print\_ex5(check->right, level - 1, t, OgLevel);

}

}

void scan\_and\_do\_ex5()

{

int num;

tree\_ptr t = buildTree();

scanf("%d", &num);

while (num != 0)

{

if (find\_node(t, num) == NULL)

{

insertTree(&t, num);

}

scanf("%d", &num);

}

int level;

scanf("%d", &level);

print\_ex5(t, level, t, level);

}

סעיף 6

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

typedef struct tree

{

int info;

tree\* left;

tree\* right;

}treeRec, \* tree\_ptr;

//stack actions

typedef struct node\_type

{

tree\* info;

node\_type\* next;

};

typedef struct stackAVL

{

node\_type\* top;

};

void init(stackAVL\* s)

{

s->top = NULL;

}

void push(stackAVL\* s, tree\* x)

{

struct node\_type\* temp = (struct node\_type\*)malloc(sizeof(struct node\_type));

temp->info = x;

temp->next = s->top;

s->top = temp;

}

int isEmpty(stackAVL s)

{

return !s.top;

}

int isFull(stackAVL s)

{

return 0;

}

tree\* pop(stackAVL\* s)

{

struct node\_type\* temp = s->top;

tree\* data ;

data = s->top->info;

s->top = temp->next;

free(temp);

return data;

}

tree\* top(stackAVL s)

{

return s.top->info;;

}

//end

int fmax(int x, int y)

{

return(x > y ? x : y);

}

int height(tree\* root)

{

if (root)

{

return 1 + (int)fmax(height(root->left), height(root->right));

}

return -1;

}

int balanceFactor(tree\* root)

{

if (root)

{

return height(root->left) - height(root->right);

}

return 0;

}

void insertTree(tree\*\* root, int value)

{

tree\* v = (tree\*)malloc(sizeof(tree));

v->right = NULL;

v->left = NULL;

v->info = value;

stackAVL st;

init(&st);

tree\* ptr = \*root, \* last;

if (!(\*root))

{

\*root = v;

return;

}

last = ptr;

while (ptr)

{

last = ptr;

push(&st, last);

if (value < ptr->info)

{

ptr = ptr->left;

}

else

{

ptr = ptr->right;

}

}

if (value < last->info)

{

last->left = v;

}

else

{

last->right = v;

}

ptr = v;

int bf = 0;

int sbf = 0; // sbf stands for son's balance factor

tree\* par;

while (!isEmpty(st))

{

ptr = pop(&st);

bf = balanceFactor(ptr);

if (bf < -1)

{ // rr rotation

sbf = balanceFactor(ptr->right);

if (sbf == -1)

{

if (!isEmpty(st))

{

par = pop(&st);

if (par->left == ptr)

{

RR\_rotate(&ptr);

par->left = ptr;

}

else

{

RR\_rotate(&ptr);

par->right = ptr;

}

}

else

{

RR\_rotate(root);

}

}

else // only RL

{

if (!isEmpty(st))

{

par = pop(&st);

if (par->left == ptr)

{

RL\_rotate(&ptr);

par->left = ptr;

}

else

{

RL\_rotate(&ptr);

par->right = ptr;

}

}

else

{

RL\_rotate(root);

}

}

}

else if (bf > 1)

{

sbf = balanceFactor(ptr->left);

if (sbf == 1)

{

if (!isEmpty(st))

{

par = pop(&st);

if (par->left == ptr)

{

LL\_rotate(&ptr);

par->left = ptr;

}

else

{

LL\_rotate(&ptr);

par->right = ptr;

}

}

else

{

LL\_rotate(root);

}

}

else // only LR

{

if (!isEmpty(st))

{

par = pop(&st);

if (par->left == ptr)

{

LR\_rotate(&ptr);

par->left = ptr;

}

else

{

LR\_rotate(&ptr);

par->right = ptr;

}

}

else

{

LR\_rotate(root);

}

}

}

}

}

tree\* find(tree\* root, int value)

{

if (!root)

{

return NULL;

}

if (root->info == value)

{

return root;

}

if (value < root->info)

{

return find(root->left, value);

}

return find(root->right, value);

}

tree\* getAVL()

{

tree\* root = NULL;

int num = 0;

do {

puts("enter a non-zero number, or enter 0 to stop:");

scanf("%d", &num);

if (num && !find(root, num))

{

insertTree(&root, num);

}

} while (num);

return root;

}

void LL\_rotate(tree\*\* root)

{

tree\* a, \* b;

a = \*root;

b = a->left;

a->left = b->right;

b->right = a;

\*root = b;

}

void LR\_rotate(tree\*\* root)

{

tree\* a, \* b, \* c;

a = \*root;

b = a->left;

c = b->right;

a->left = c->right;

b->right = c->left;

c->right = a;

c->left = b;

(\*root) = c;

}

void RL\_rotate(tree\*\* root)

{

tree\* a, \* b, \* c;

a = \*root;

b = a->right;

c = b->left;

a->right = c->left;

b->left = c->right;

c->left = a;

c->right = b;

(\*root) = c;

}

void RR\_rotate(tree\*\* root)

{

tree\* a, \* b;

a = \*root;

b = a->right;

a->right = b->left;

b->left = a;

\*root = b;

}

void main()

{

}